# PermNet: Permuted Convolutional Neural Network

**A THESIS**
**SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL**
**OF THE UNIVERSITY OF MINNESOTA**
**BY**

**Rishabh Mehta**

**IN PARTIAL FULFILLMENT OF THE REQUIREMENTS**
**FOR THE DEGREE OF**
**MASTER OF SCIENCE**

**Ju Sun, Zhi-Li Zhang**

**May, 2021**

# Acknowledgements

I would like to thank my advisors, Professor Ju Sun and Professor Zhi-Li Zhang for their continued technical guidance and for bestowing the motivation throughout the project period. I would also like to thank my committee member Professor Changhyun Choi for his help in thesis reviewing process. A special shoutout to Professor Ju Sun for uttering the words "permutation" during some technical discussion, which sparked the idea of permuted convolutions, as well as for helping me come up with the exact formulation. I would also like to thank all my friends in my academic life for bearing with my constant barrage of presentations and technical discussions with them and giving me helpful advice.

# Dedication

To my parents, my sister, my friends and the wonderful advisors at UMN for motivating me to stay on top of the hectic academic life and for supporting my attempt to pursue risky, yet rewarding project for thesis.

## Abstract

Convolution filters in CNNs extract patterns from input by aggregating information across height, width and channel dimensions. Information aggregation across height and width dimensions performed using depthwise convolution, helps identify neighborhood patterns and hence is very intuitive. However the method in which channel dimension information is aggregated by channel summation seems mathematically simplistic and out of convenience. In this project we attempt to improve the channel dimension aggregation operations. The first approach introduces weighted summation channel aggregation in convolutions. The second approach introduces permuted convolutions which attempt to perform psuedo-width scaling by generating new constrained filters from existing filters. Implementing permuted convolutions comes with many challenges such as permutation explosion, stochasticity, higher memory and computation requirements. To resolve these issues, we come up with multiple variants of permuted convolutions and present their advantages and disadvantages. Lastly, we provide empirical results showcasing the performance of weighted channel summation networks and permuted convolution networks, present our findings and recommendations for future work.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The field of Computer Vision has taken a huge leap since the AlexNet [3] architecture proposed for ImageNet 2012 challenge [4]. By achieving the state of the art performance, AlexNet authors showcased the advantages of convolution based neural architectures and brought Convolutional Neural Networks (CNNs) to the limelight. After that, every first winner architecture since 2012 on ImageNet challenge [4] has been based on CNN. Researchers started experimenting with deeper and high capacity CNNs, such as VGG [5] and Inception [6] architectures. There have been many ideas proposed to improve the learning power of deep CNN since. In 2016, the winning ResNet [7] architecture introduced the concept of residual connections which provide highways for gradients during backpropogation and control the complexity of the network by allowing networks to learn identity mapping. The DesneNet [8, 9] architecture later built on the ResNet [7] architecture by extending residual connection highways from any layer to all of its succeeding layers inside the DenseBlock. The EfficientNet [1] architectures proposed by Google in 2019 provided deeper understanding into scaling different dimensions of CNN and achieved state of the art results on then discontinued Imagenet challenge.

As the power of deep CNNs started being realized, researchers started creating scaled up versions of CNNs for various vision tasks, which seem to outperform their shallow counterparts. On the other hand, some researchers have been focusing on taking the other side of trend, intending to create lower complexity CNNs that can be run on resource constrained devices such as edge devices and mobile phones. The field of network compression, creating low memory and computation requiring models without

significant drop in model accuracy, has taken off in recent years as tech industry giants attempt to bring the power of deep neural networks to mobile phones and edge devices to perform various tasks with better quality and performance. Some seminal architectures proposed in the field such as MobileNet [10, 11] and SqueezeNet [12] have brought about a revolution in edge computing. Today people use various network compression techniques such as model pruning [13, 14], quantization [15, 16, 13] and parallelism [15] to create very compact CNNs with good accuracy and performance on the task at hand. This progress can be seen in all kinds of visual prediction tasks such as super resolution, object detection, object classification and object tracking. Taking the case of single image super resolution, the state of the art networks in the field are tens of millions of parameters large [17, 18, 19]. However, recent progress in network compression has led to creation of very compact CNNs with few hundred thousand parameters that perform almost equally well as state of the art architectures [20].

### 1.0.1 Introduction to network scaling

The creation of higher complexity or lower complexity CNNs by changing the dimensions of the baseline network is called network scaling. As EfficientNet [1] paper points it out, There are three different dimensions through which networks can be scaled up or down. These dimensions are network depth, network width and input resolution. This dimension scaling method can be visualized in Figure 1.1, where baseline network has been scaled up in three different dimensions. Network depth can be scaled by increasing the number of layers of CNN. Network width is scaled by increasing the number of convolution filters in each individual convolution layer. Input resolution can be scaled by using the higher resolution images as input. All three upscaling dimensions help increase network complexity. The increase in network complexity can be understood in two terms - increase in number of learnable parameters of the network and increase in computation, signified by number of floating point operations (FLOP) of the network. The number of parameters of a fully convolutional network scale linearly with the network depth and in a squared relationship with the network width. The number of parameters aren't affected by the input resolution. However the FLOP increase in a squared relationship with the scale of input resolution. However in most computer vision tasks, we do not have higher resolution input images available and hence height and width of the network

are the two most important model scaling dimensions.

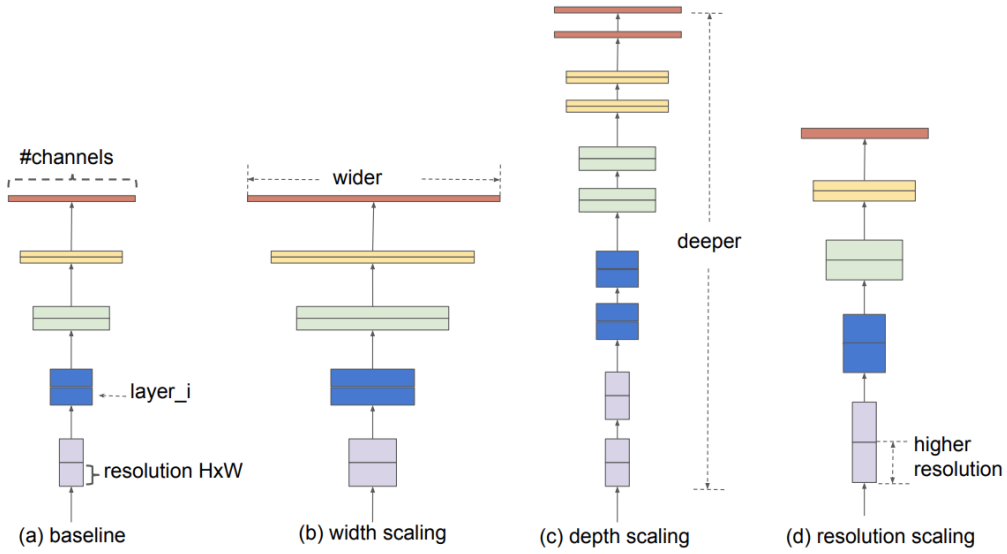


Figure 1.1: Model scaling. (a) The baseline network to be scaled. (b-d) Networks with width, depth and input image resolution scaled respectively. Source: EfficientNet[1]

The important thing to note here is that the number of parameters are most affected by increase in width of the network due to the squared proportionality. Hence width scaling helps improve network capacity, however at the cost of parameter increase. With permuted convolutions, it is possible to generate additional filters, which share the weights with base filters in the network without any additional parameter costs. Since we only simulate the additional filters without adding any new weights in the network, we call this process as psuedo-width scaling. Psuedo-width scaling might help increase the network capacity passively. However this needs to be proven and we perform empirical comparisons to identify if this is true. In this paper we intend to compare PermNets with their corresponding CNN counterparts and understand their learning ability differences. We focus on the image classification problem for the scope of this paper and perform all experiments on the popular CIFAR-10 [21] dataset.

The main contributions of this project are as follows:

- Proposing a change to convolution layer with weighted summation of filter channels instead of direct summation performed in typical convolution.

- Proposal of permuted convolutions to simulate width scaling effect, as well as discussion of various architectures of networks employing permuted convolutions - called PermNets.

The further chapters have been divided as follows:

- Chapter 2 introduces various convolution types that will be used to come up with permuted convolution formulation.
- Chapter 3 covers the related work done in this field. In this section we will talk about various architectures proposed for making networks more compact and suited for mobile devices as well as discuss an architecture that is closest to the PermNet.
- Chapter 4 details the problem statement and the motivation behind it.
- In Chapter 5 discusses in detail the architectures of weighted channel summation network (WeightedNet) and PermNet variants.
- Chapter 6 describes the empirical results of the experiments performed to understand the performance of PermNet and WeightedNet.
- Chapter 7 presents our findings and recommendations about training PermNet.
- Chapter 8 discusses the future work we consider important to make PermNet practical.
- Chapter 9 discusses our conclusion about the project.

# Chapter 2

# Convolution Background

The purpose of performing convolution is to extract useful features from the input. In Convolutional Neural Network, different features are extracted through convolution using filters whose weights are automatically learned during training. There have been many variants of convolution proposed in the image processing community. We will discuss the ones that are relevant to understand permuted convolution architecture.

### Normal Convolution

The typical convolution in CNN extracts features from height, width and channel dimension from input. However this process can be divided into two processes - pattern extraction across height and width dimension, and channel summation across depth dimension. During the convolution process, rigorously speaking, cross correlation is performed between each channel of input and corresponding channel of convolution filter. The cross-correlation output contains extracted features from height and width dimensions of the input. In the next step, the output of cross-correlation from each channel is then summed up across the channel dimension to obtain the final output of the convolution filter. Hence through this channel summation phase, we aggregated extracted patterns by cross-correlation and obtain the final pattern extracted from height, width and channel dimension by the convolution filter. Figure 1.2 highlights the typical convolution process. The number of parameters of a typicaal convolution layer would be equal to multiplication of number of input channels, filter height, filter width and

number of filters in the layer.

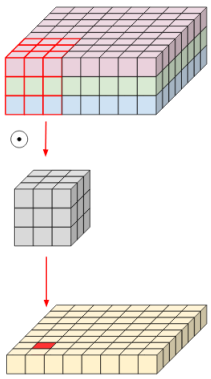

Figure 2.1: Typical convolution in CNN. Source: EfficientNet[1]

### Depthwise convolution

Depthwise convolution is a sub-part of typical convolution. In depthwise convolution, each channel of input is convolved with each channel of filter. Hence, each 2d surface from both input and filter is convolved separately. Finally the output of depthwise convolution is the concatenation of all convolved channels. For each filter, the number of output channels through depthwise convolution are the same as number of input channels. Depthwise convolution only extracts patterns from height and width dimensions. However it does not aggregate these patterns through channel summation. Figure 1.3 represents the depthwise convolution operation. The number of parameters of depthwise convolution are the same as number of parameter of normal convolution.

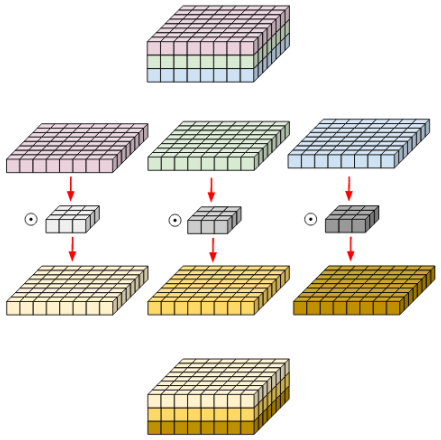

Figure 2.2: Depthwise convolution

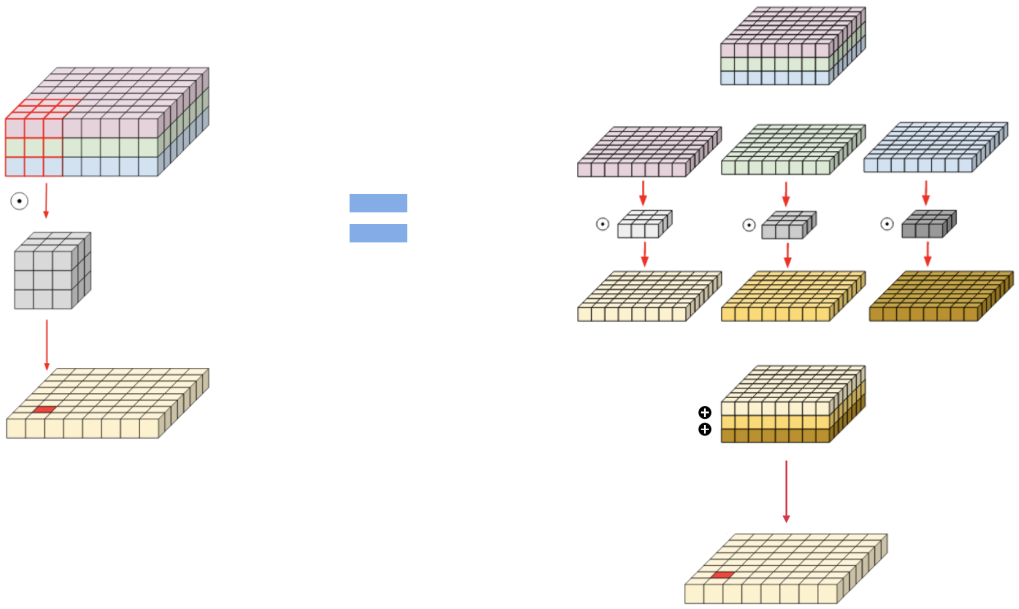

Figure 2.3: Relationship between normal convolution and depthwise convolution

An intuitive relationship between depthwise and normal convolution is that normal convolution first performs depthwise convolution and then performs channel summation. This is evident from Figure 1.4, where we can see that by summing up the channels that are generated as output by depthwise convolution, we obtain the result of normal

convolution. Hence depthwise separable convolution is, simply put, normal convolution without the channel summation process.

### Depthwise separable convolution

Depthwise separable convolution were popularized by Xception [22] and MobileNet [10, 11] papers. Depthwise separable convolution work in two stages. In the first stage, depthwise convolution is performed, with only one 3d filter. In the second stage, we aggregate the channels of depthwise convolved output using point-wise convolutions, also called separable convolution. Point wise convolutions make use of filters of height and width 1. Intuitively, pointwise convolution perform weighted summation across channel dimension. In its typical formulation, depthwise separable convolution makes use of only one FxF filter, where F is greater than 1. This FxF filter is used during depthwise convolution. Then multiple point-wise convolution filters are utilized to generate final output channels. The advantage of using depthwise separable convolution is that it can help reduce the parameters of a convolution layer significantly. This reduction in parameter count occurs due to the fact that we offload the output channel generation task on point-wise convolution instead of keeping it on FxF convolution. Point-wise convolution has filter size 1, and hence its parameter count is much smaller than that of the normal convolution, given the other criteria stay the same. Due to its lower parameter count, depthwise separable convolutions have become very popular in the domain of network compression and for building networks that run on resource-constrained devices.

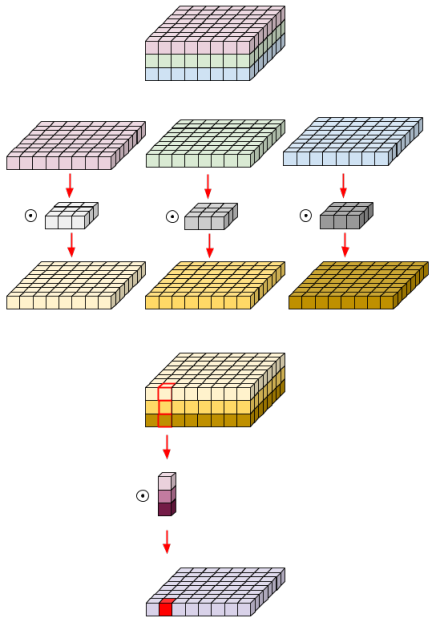

Figure 2.4: Depthwise separable convolution

### Grouped convolution

First proposed by authors of AlexNet [3] for allowing network to be trained on two separate GPUs in parallel, grouped convolution provides parallel path for convolution layer computation. In grouped convolution, both input and filters are divided in individual groups that are then convolved separately. The output of various groups are then concatenated together to obtain the final output. Grouped convolution can help reduce the parameters of the convolution layer. The number of parameters of the layer are inversely proportional to number of groups. The reduction in parameter comes due to the fact that input and output channels for each group get reduced by number of groups generated. The other advantage of grouped convolution is efficient learning. Since the convolutions are divided into several paths, each path can be handled separately by different GPUs in parallel. However one big disadvantage of grouped convolution is that the information across channel dimension in separate groups is not aggregated by the network. Each filter group only handles information passed down from the fixed portion in the previous layers. For examples in Figure 1.7, on the left, the first filter

group (red) only process information that is passed down from the first 1/3 of the input channels. Similar story occurs for green and blue groups. As such, each filter group is only limited to learning a few specific features. This property blocks information flow between channel groups and weakens representations during training. To overcome this problem, we apply the channel shuffle.

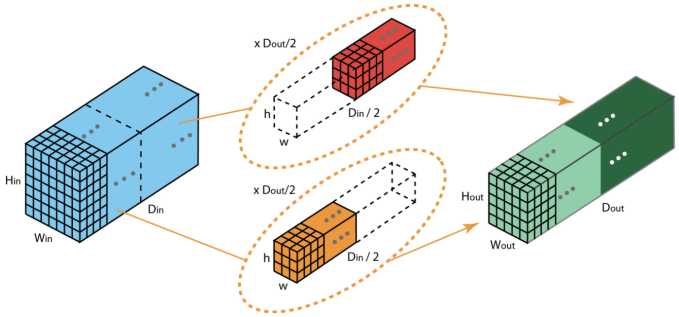


Figure 2.5: Grouped convolution

**Shuffled grouped convolution**

To improve upon the grouped convolution, the idea of channel shuffle is that we want to mix up the information from different filter groups. In Figure 1.7, on the right, the feature map is obtained after applying the first grouped convolution GConv1 with 3 filter groups. Before feeding this feature map into the second grouped convolution, the channels in each group are divided into several subgroups. Then these subgroups are mixed up.By performing channel shuffling operation, information can now flow between channel groups, increasing the representation of input during training. The ShuffleNet [2] authors have adopted shuffled grouped convolution for their architecture and popularized the method since. We will take inspiration from the idea of channel shuffling to come up with permuted convolution formulation.

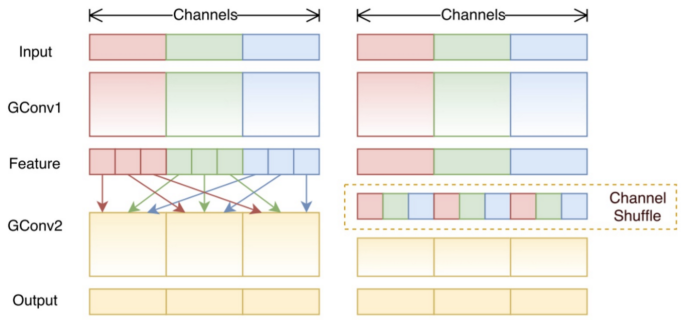


Figure 2.6: Channel shuffling approach to counter drawbacks of grouped convolution. Source: ShuffleNet [2]

# Chapter 3

# Related Work

Since AlexNet paper popularized CNN, convolution layers have become primary modules to extract patterns from visual data. As network sizes kept increasing, a new field of network compression started forming to allow these state of the art CNNs to run on resource constrained devices. There have been many approaches published for building compact models. The most popular and novel methods have been proposed in MobileNet [10, 11], Xception [22] and SqueezeNet [12] papers. The Xception [22] paper popularized the concept of depthwise separable convolution, which can help reduce convolution parameters. As discussed before, depthwise separable convolution can help reduce network parameters by offloading more computation to cheaper 1x1 convolutions. The Squeezenet [12] paper makes heavy use of 1x1 filters to reduce channels in the network, keeping parameter count low. The MobileNet [10, 11] papers build on this idea by performing depthwise separable convolution in all of the residual blocks in the network.

Another related approach that has motivated this work is introduced in the ShuffleNet [2] paper, where output of filter channels are shuffled in order to alleviate drawbacks of grouped convolution. The shuffled grouped convolution operation used by ShuffleNet can be seen visually in Figure 2.6. An important thing to note here is that although the channel shuffling operation in ShuffleNet seems stochastic, it actually is deterministic. Hence there is not stochasticity introduced in the network due to shuffling operation. The current project builds on the ideas of depthwise convolution and channel shuffling introduced in MobileNet [10, 11] and Shufflenet [2] papers respectively.

# Chapter 4

# Problem Statement and Motivation

This primary objective of this project is to expand the capacity of CNN by performing pseudo width scaling without introducing any new parameters in the network. We also want to change the convolution operation in CNNs and bring in mathematically more sound operations to it, especially during channel information aggregation step. The exact task we tackle in this paper is to attempt to improve upon the accuracy of state of the art image classification CNNs by permuting convolution filter channels to generate additional constrained filters. The intuition behind being able to achieve these objectives is that permuted convolutions can extract more complex patterns from input images than typical convolution operations do by permitting sharing of extracted information across filters.

Recent state of the art CNNs have large parameters count, mostly due to large number of convolution filters. There has been an ever increasing interest to make neural models work on mobile phones and edge devices. Since these devices are resource-constrained, in terms of both memory and computation, neural model compression is of fundamental importance to make large CNNs work on such devices. If we can scale up the small network using permuted convolutions, it might be able to achieve similar accuracy as a larger network. And in this case, we have been able to achieve model compression on the larger network with the help of permuted convolutions. Hence,

model compression has served as a good motive to us when coming up with permuted convolution formulation.

The other motive behind using permuted convolutions is more technical. As we can see in Figure 2.3, depthwise convolution along with channel summation is equivalent to performing normal convolution. If we leave the channel summation part out for now and only focus on depthwise convolution output, we see that depthwise convolution has only extracted patterns across height and width dimensions from input. Consider an RGB input to the convolution layer. Then, after depthwise convolution with multiple filters, we can see that we have extracted patterns from R, G and B channels separately with each filters. The next step is to aggregate the information across channel dimension. However, now some interesting opportunities come up. Considering that each channel of each filter convolved depthwise with the input separately, all of the channels in output of depthwise convolution are independent. Now the question arises that which R, G and B convolved channel should we aggregate. The simplistic answer would be that we aggregate convolved R, G and B channels of each filter. However, upon careful thought it becomes clear that we can aggregate any convolved R, G and B channels without having to impose that these channels come from the same filter. Effectively, we should be able to aggregate convolved R, G and B channels from separate filters, since as we established before, all of the channels in depthwise convolution output are independent. Hence, there need not be any constraint that we can only aggregate convolved channels of each filter. We should be able to aggregate convolved channels across different filters. This deduction serves as the intuition behind permuted convolutions and PermNet.

The motive for WeightedNet is to counter mathematical simplicity that exists in channel aggregation phase of convolution. As evident from Figure 2.1 and Figure 2.3, normal convolution is equalivalent to performing depthwise convolution and then channel summation across depthwise convolved output. While the intuition behind depthwise convolution is to extract patterns across height and width dimensions, the exact intuition behind channel summation is unclear. While we would like to aggregate patterns across channel dimension in order to account for correlation in patterns among channels, the summation operations seems to be applied due to its simplicity. By summing up information across channels, we are introducing our bias that pattern extracted across each channel is equally important. However this assumption is not supported by any

observation or evidence. When training neural networks, it is better to let the network figure out such operations with the help of backpropagation rather than us imposing strict rules on them. This serves as our intuition behind replacing channel summation operation with learnable filters and create the weighted convolutions. While weighted convolution may sound similar to depthwise separable convolution, there are key differences between them that we will discuss in later sections.

# Chapter 5

# Methodology

As discussed in motivation section, there is an opportunity to modify the channel summation procedure in normal convolution. We propose multiple different networks to achieve this. The WeightedNet network that we propose introduces weighted summation across channels in convolution. Then, deriving upon the idea of permuted convolutions, we propose PermNet. We also propose various variants of PermNet that attempt to address certain disadvantages of the PermNet architecture.

### 5.0.1 WeightedNet

Normal convolution operation can be divided in two steps. The first step is the extraction of patterns across height and width dimension. The second step is to perform channel summation across depth dimension. WeightedNet replaces this second step with weighted summation across channel dimension instead of direct summation. As discussed in motivation section, direct summation of filter channels is utilized in normal convolution for simplicity and faster computation. However direct summation operation introduces the bias from our end that each convolved output channel of filter is equally important in extracting useful patterns. However the basis for such assumption is mathematical simplicity and it has not been proven to be true. We believe that instead of introducing such bias ourselves, we can let the network figure out what the weight of each channel should be. Hence, taking inspiration from depthwise separable convolution, we introduce weighted convolutions.

We implement weighted convolution with the help of depthwise convolution and grouped 1x1 convolution, with number of groups equal to number of $FxF$, where $F > 1$, filters present in the layer. Here the task of grouped 1x1 convolution is to perform weighted summation aggregation across channel dimension for each filter separately and concatenate the results. The 1x1 grouped convolution has learnable parameters and the motivation behind using as it is purely implementation ease and better performance with machine learning libraries, such as Pytorch, than other possible approaches.

While the inspiration for weighted convolutions has been taken from depthwise separable convolution, they differ quite a bit in their actual formulation. In its most common form depthwise separable convolution uses only one $FxF$ filter, with $F > 1$. Most of the responsibilities to generate output channels fall on multiple 1x1 separable convolutions. In the case of weighted convolution, there can be multiple $FxF$ filters, with $F > 1$. Most of the computational responsibilities still lie on $FxF$ filters. Also, instead of using multiple 1x1 filters, we only use one filter that is divided in groups. The number of groups would be equal to number of $FxF$ filters in the convolution layer.

However, in its most general form, depthwise separable convolution can have multiple $FxF$ filters, with $F > 1$. In this case, multiple 1x1 separable filters are used in order to generate final output. However, the while many similarities exist between such depthwise separable convolution and weighted convolution, the primary difference still remains. This difference is that weighted convolution network uses grouped 1x1 convolution to generate output channels, while depthwise separable convolution uses normal 1x1 convolution filters.

The intuitive difference between depthwise separable convolution and weighted convolution is that we try to generate additional linear filters from a base filter in the former case, while we try to perform weighted summation across channel dimension in the later case. Hence the primary aim of the depthwise separable convolution is to reduce number of parameters of the network, and for weighted convolution the aim is to perform better pattern aggregation across channel dimension.

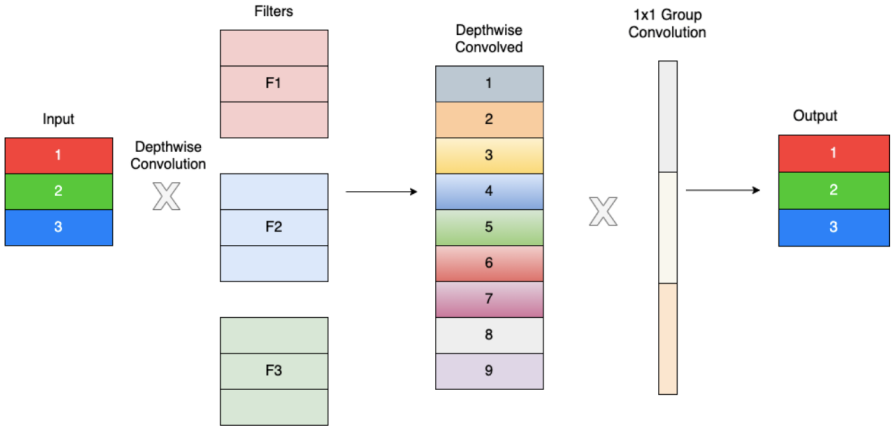


Figure 5.1: An example of weighted convolution in WeightedNet.

### 5.0.2 PermNet

We propose PermNet architecture that uses permuted convolution layers in place of typical convolution layers. We will talk about the idea behind permuted convolutions as well as discuss various architectures that implement it.

#### Permuted Convolution

Consider an input image with R, G and B channels. Consider passing this image to a convolution layer with multiple $FxF$, where $F > 1$ filters. Let's consider breaking down convolution operation here. The first phase would be to extract patterns across height and width dimensions from image. We can achieve this using depthwise convolution. Now we would have convolved R, G and B channels coming from each of the filter in the layer. Now the question arises, if we can aggregate these channels in a novel manner. We know that each channel of each filter was convolved independently while performing depthwise convolution. We can aggregate the R channel convolved output of one filter with B and G channel convolved outputs from other filters, since all of these channels were convolved separately. Hence, there is no reason for us to only aggregate convolved R, G and B channels of each filter themselves. We can mix-and-match the convolved R, G and B channels across filters. When we aggregate outputs of different channels of different filters, we generate a permutation of aggregated channels. There can be

many such possible permutations in a convolution layer. Each new permutation that we generate, is effectively simulating a new filter, whose channels come from existing filters. We call this new filter as constrained-filter. It is called such due to the fact that such filter has the constraint that all of its channels come from existing filters in the convolution layer, that we call base filters. Hence constrained filters do not have their own weights and do not contribute to parameter increase. However since we created new filters out of base filters and can utilize them in the network, we have effectively scaled the network width. Since this scaled network width is actually a simulated width scaling, we called it pseudo-width scaling.

**Stochastic PermNet**

Stochastic PermNets are the variants of PermNet that have stochasticity in their architecture. We use random permutation with certain constraints to implement permuted convolutions in these variants. The problem with such architectures is that at inference time we would sample a permutation again, which may not be reflective of all the permutations learned by the network. However, the learned network during training would be the average of all the permutations explored during training. The inference time permutation selection problem introduced by stochastic PermNet is primary reason why we would search for deterministic architectures next.

The stochastic variants will attempt to generate new permutations iteratively and the final network will be the aggregate of all the permutations explored. We will now discuss the three variants of stochastic PermNet.

- PermIterWeightedNet:

  The PermIterWeightedNet generates a newly permuted constrained filter at every iteration during training. Similarly during evaluation phase, it generates a constrained filter for each testing batch. A permuted constrained filter is generated in three steps. The first step is to perform depthwise convolution on input x with all n filters to be used in a convolution layer. Let its output be depthwise(x), thus extracting information across height and width dimensions. The second step is to perform random permutation on the output of depthwise convolution depthwise(x). Albeit, this random permutation process follows some constraints that

are discussed in the subsection below. Let the permuted output be permuted(x). The third step then is to combine channels from permuted(x) using 1x1 filters, hence extracting information in channel dimension. The 1x1 convolutions here are grouped. Grouped convolutions were first introduced in AlexNet paper [3]. The number of group would be equal to the number of channels of input x. The output of 1x1 convolution represents the output of permuted convolution layer.

- Constraints for permuted convolutions: The constraint imposed on random permutations is necessary to make sure the order of channels in generated constrained filter matches that of original convolution filter. More precisely, $i^{th}$ channel of generated constrained filter should be obtained from $i^{th}$ channel of one of the original convolution filters.
- Permutation explosion: The total number of permutations possible in a convolution layer with n convolution filters, with each filter of depth d equals to $[n + \frac{k*(n-1)*n}{2}]$. As one can see, this figure is of the order $O(n^2)$, compared to $O(n)$ for typical CNN convolution layer. As PermIterWeightedNet becomes deeper, the number of total permutations possible in the PermIterWeightedNet grows rapidly. This problem arises due to the fact that total number of permutations of PermIterWeightedNet is the multiplication of number of permutations of each convolution layer in its architecture. In a typical ResNet-18 block where there are two convolution filters with 64 filters each, if we permute all channels of filter the number of permutations of that residual block crosses 16 million. And since there are four such residual blocks in ResNet-18, the total permutations possible in ResNet-18 is over 65 billion. For ResNet-50, the total number of permutations is unimaginably high. There is no way we could train such large networks while exhausting every possible permutation.
- BoundedPermNet: Bounded permuted convolutions. To counter the permutation explosion problem, there needs to be set an upper bound on total number of permutations performed in PermIterWeightedNet. There are multiple ways to achieve this. They are discussed below.

- Bound on channels permuted: In this approach only specific number of channels are permuted. This approach consumes no additional memory, however some channels which are permuted might be construed by network differently, either more important to learn or less important to learn. Understanding the effects of such bounded permutation is part of the future work.
- Bound on total permutations: In this approach, all the channels of convolution filters are permuted, albeit an upper limit is set on total number of permutations allowed for each layer. Upon crossing the limit, the network would regenerate permutations that have already been explored. Additional memory is required to store the permutations explored before hitting the upper limit. The memory required is directly in proportional to the sum of number of permutations allowed in each layer. Implementing this approach is part of the future work.

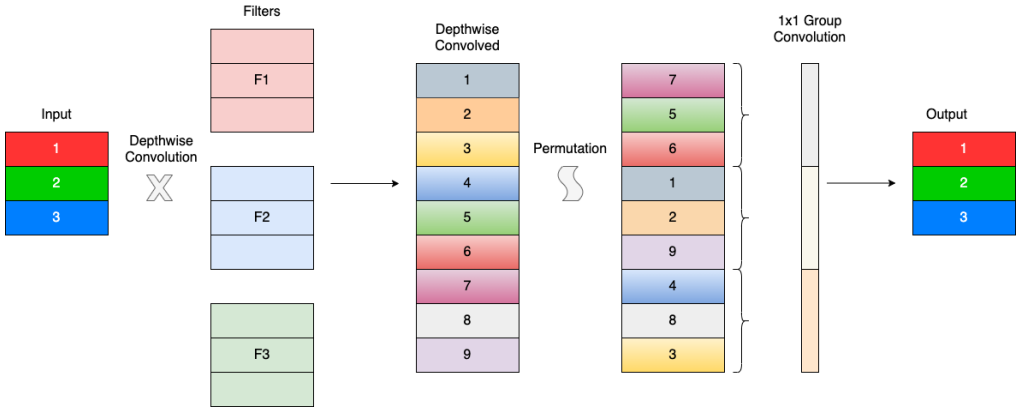


Figure 5.2: PermIterWeightedNet: Permuted convolution visualized. The convolution filters are depthwise convolved with the input. The output channels of depthwise convolution are then randomly shuffled with constraints. The figure showcases one such shuffling scenario with shuffled channel numbers. The final output is obtained by performing 1x1 group convolution with number of groups equal to number of convolution filters used during depthwise convolution.

- PermIterNet:

The PermIterNet has similar architecture to PermIterWeightedNet, except for a minor change. PermIterNet uses direct summation across filter channels instead of weighted summation using 1x1 convolution in PermIterWeightedNet. The direct summation approach helps reduce parameters further while reducing the complexity of the network. The architecture of PermIterNet is shown in Figure below.

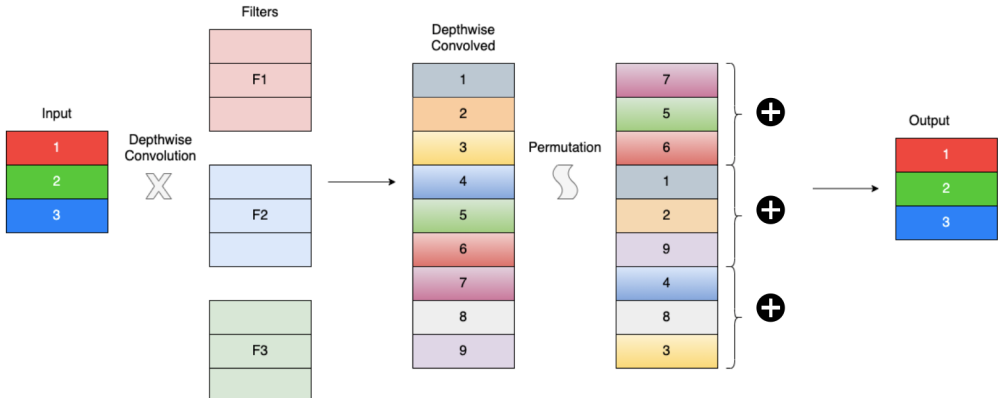


Figure 5.3: PermIterNet: Permuted convolution visualized. The convolution filters are depthwise convolved with the input. The output channels of depthwise convolution are then randomly shuffled with constraints. The figure showcases one such shuffling scenario with shuffled channel numbers. The final output is obtained by summing up channels for each filter in the layer.

- PermShuffleNet: This approach is one simple variation of PermNet. In this approach no additional constrained filters are generated. However instead of permuting individual channels of filters, entire filters are permuted. Hence, at every iteration, a different arrangement of original filters is obtained. This approach is different than ShuffleNet [2] since the ShuffleNet architecture uses group convolutions in general and rearranges specific channels across groups after convolution operation. While PermShuffleNet evidently shuffles all channels randomly without any restrictions and does not work just in the context of group convolutions. BoundedPermShuffleNet architecture follows similar logic as BoundedPermNet and shuffles only a specific number of filters to generate a new arrangement of filters.

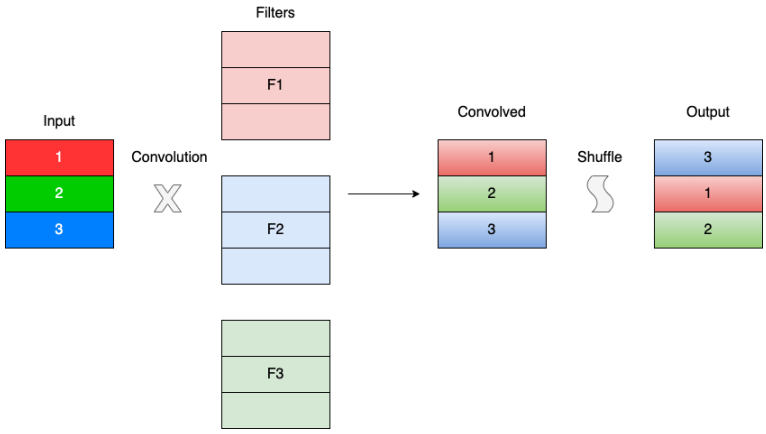


Figure 5.4: PermShuffleNet: Shuffled convolution visualized. The filters are convolved with the input in typical convolution fashion. The convolved output channels are then randomly shuffled to obtain the final output of shuffled convolution layer. The figure showcases one such shuffling scenario with shuffled channel numbers.

### Deterministic PermNet

As seen with stochastic PermNet architectures, performing permutations with the help of a random variable can lead to difficult inference time permutation choices. The stochasticity was introduced so that we can generate new permutations iteratively. However, if we can figure out the ways to generate permutations without introducing stochasticity in the network, we can create deterministic networks that still benefit from permuted convolutions.

We will now discuss three different deterministic architectures that we experimented with for implementing permuted convolutions.

- PermDeterWeightedNet: PermDeterWeightedNet is a deterministic PermNet architecture. For this architecture, we generate some random permutations initially but then later fix them. To improve the learning capability of the network, we still keep our base filters, but simply add additional constrained filters in the network. These additional filters would be kept fixed throughout the training duration of the network. The number of constrained filters to generate is a hyperparameter. For our experiments, we keep the number of constrained filters equal to number

of base filters. Hence, we have increased width to 2x the initial width with the help of fixed permutations.

The next step after generating the immutable permutations, the next step would be to aggregate channels of each filter. For this architecture, we go with weighted summation approach for filter channels with 1x1 grouped convolution, taking inspiration from WeightedNet. However, since we have increased the width of our network, the number of output channels after 1x1 grouped convolution would be more than the number of output channels we obtain with baseline network. To keep comparison fair, we should have the same number of output channels in both baseline and our competing network. To reduce the number of output channels, pointwise convolutions are used, which are a popular way in community to achieve channel reduction. The convolution layer of PermDeterWeightedNet can be visualized in Figure 5.5.

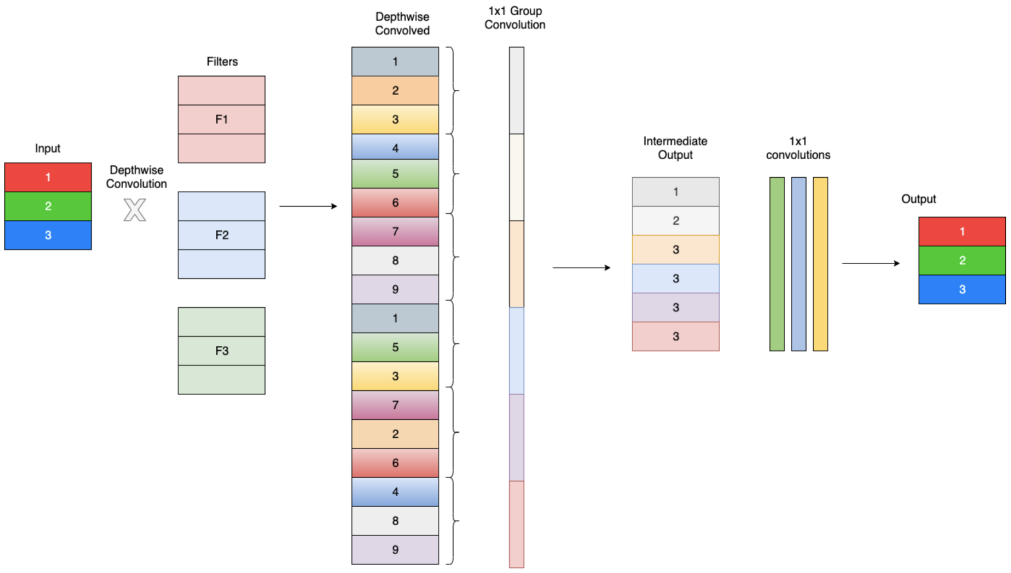


Figure 5.5: PermDeterWeightedNet: Deterministic unmutable permuted convolutions, with weighted summation across filter channels.

- PermDeterNet: PermDeterNet is just another variant of PermDeterWeightedNet. In this case, the only difference is that we replace the weighted channel summation

of filter channels in PermDeterWeightedNet to direct summation. PermDeterNet is just an extreme case of PermDeterWeightedNet, where all of weights of 1x1 group convolution are equal to 1. The other modules of the architecture are the same as PermDeterWeightedNet.

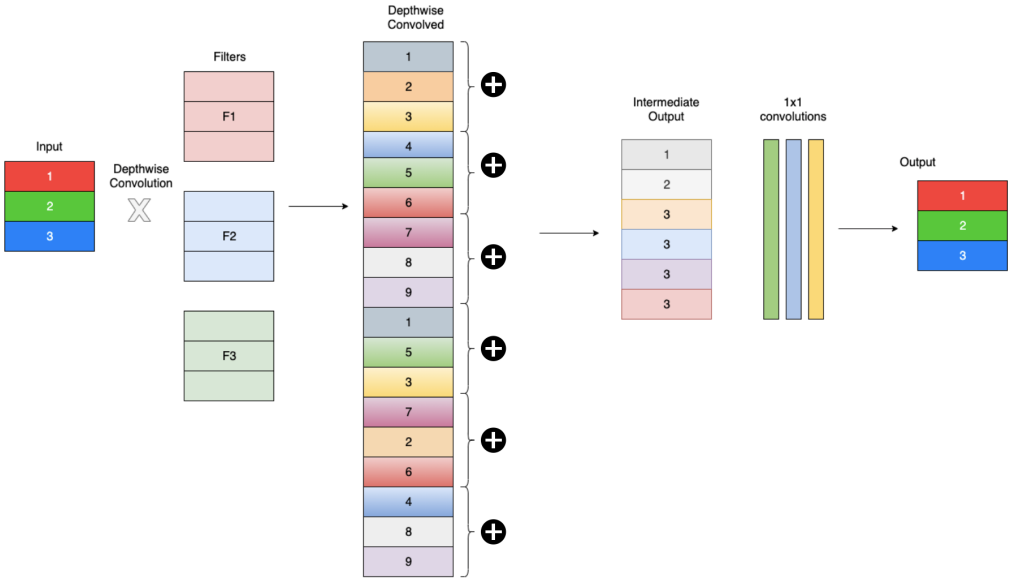


Figure 5.6: PermDeterNet: Deterministic unmutable permuted convolutions, with typical summation across filter channels.

- PermAutoMultiNet: The deterministic architectures we discussed till now fix the permutations while training. These permutations are chosen randomly initially. However choosing random permutations introduces bias on our part that the chosen permutations would perform well. In PermAutoMultiNet, we let the network figure out which permutations it wants to use. We make this process completely deterministic with the help of imposed constraints on the network. We first perform depthwise convolution with multiple filters and extract patterns. Next, we use 1x1 convolutions with the sparsity constraint. Intuitively, each 1x1 convolution we use selects a subset of channels from the depthwise convolved channels. These selected channels generate a permutation. The important thing to ensure here is that 1x1 convolution weights are sparse. We impose sparsity constraint

with the help of l1 loss function applied on the weights of 1x1 convolutions.

Hence, with the help of learnable sparse 1x1 convolutions, we can let the network learn what permutations it wants to use. The visual representation of generating such deterministic permutations has been shown in Figure 5.7.

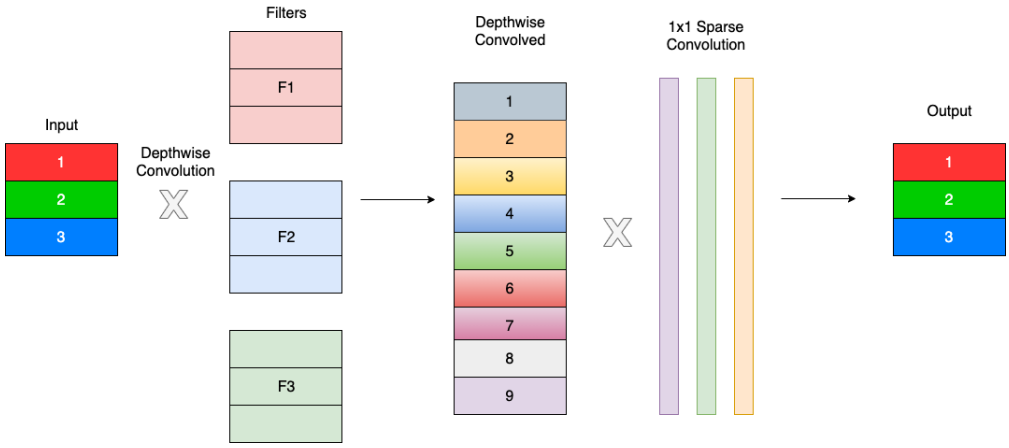


Figure 5.7: PermAutoMultiNet: Deterministic permuted convolution with learnable permutations. The sparsity constraint is imposed on multiple 1x1 convolutions with the help of l1 loss.

- PermAutoNet: This architecture is another variant of PermAutoMultiNet. A problem arising from PermAutoMultiNet is that due to 1x1 convolutions being applied on a large depthwise convolved output tensor, the number of parameters increase drastically. For context, the reduced ResNet-18 architecture we will discuss in experiments has 714,260 parameters. In comparison, PermAutoMulti ResNet-18 has parameter count of 9,102,644. Hence this is more than 12x increase in parameter count simply due to multiple 1x1 convolutions being applied on depthwise convolved output.

  We can try to reduce this parameter count by means of reducing the number of 1x1 convolutions we use for convolving with depthwise convolution output. However, then the problem becomes that we do not get the exact number of output channels we desire. To counter this, we came up with using only one 1x1 convolution, however dividing it in the number of groups equal to number of output channels

desired. Now, in cases of all the networks we would experiment with, all of them have the desired number of output channels after 1x1 convolutions equal to number of $FxF$, where $F > 1$ filters used. This works in our advantage since now we are simply performing weighted summation across filter channels for each filter, albeit with sparseness. While we cannot make a theoretical case for such convolution, we would still like to experiment with it since it helps us reduce parameter count of PermAutoMultiNet significantly.

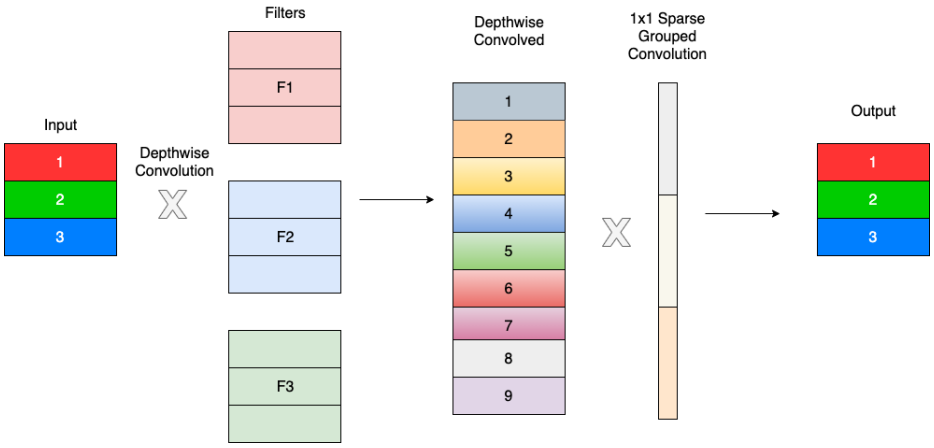


Figure 5.8: PermAutoNet: Deterministic permuted convolution with learnable permutations. The sparsity constraint is imposed on the grouped 1x1 convolutions with the help of l1 loss.

# Chapter 6

# Experiments and Analysis

In this chapter we provide the comparison between different PermNets and baseline models.

## 6.1 Dataset

We focus on the task of image classification and choose CIFAR-100 [21] as our primary dataset for experimentation. CIFAR-100 has 50k training and 10k testing images, distributed among 100 popular object classes.

## 6.2 Training & System Details

We train multiple baseline networks such as ResNet-18, ReducedResNet-18, SmallCNN and LeNet as we explore the effect of permuted convolutions on different architectures. We implemented each model using PyTorch and trained them on a GPU cluster. We train our models for 2000 epochs, unless we see them converge faster, in which case we use early stopping. The learning rates have been varied depending on models, however the most general learning rate is 1e-3. We use cross entropy loss for calculating classification error. We use SGD optimizer with momentum 0.9 and weight decay of 5e-4. For some experiments, we also use Adam optimizer without any weight decay. We also use a learning rate scheduler that reduces learning rate by a factor of 0.1 when it identifies plateau loss surface within 10 iterations. We make use of batch size

of 250 for the experiments. For PermAutoNet, we use regularization constant of 5e-3 multiplied with the l1 loss calculated on the weights of point wise convolution during loss calculation. The GPU cluster we train on contains 4 Nvidia Quadro RTX 6000 GPUs, with each GPU having 24 GB memory. However, each model was trained on a single GPU core in the cluster. We make use of CometML as our experiment logging and tracking tool.

.

## 6.3 Baseline Model

### 6.3.1 ResNet-18

We make use of ResNet-18 architecture. The model contains initial convolution, 4 layers and a linear layer at the end for classification. Each layer contains two residual blocks. Each block consists of two convolution layers, both of which are followed by batch normalization layers. The residual skip connection origins at the beginning of the block and ends at the end of the block. The number of channels in first layer is equal to 32. The number of channels are doubled in each consecutive layer. Hence, the fourth layer has 256 channels in its blocks.

### 6.3.2 Reduced ResNet-18 (RResNet-18)

We reduce the width of the ResNet-18 architecture for faster training. Similar to ResNet-18, the model contains initial convolution, 4 layers and a linear layer at the end for classification. Each layer contains two residual blocks. Each block consists of two convolution layers, both of which are followed by batch normalization layers. The residual skip connection origins at the beginning of the block and ends at the end of the block. The number of channels in first layer is equal to 16. The number of channels are doubled in each consecutive layer. Hence, the fourth layer has 128 channels in its blocks. We treat Reduced ResNet-18 as baseline and for simplicity, will call it RResNet-18 throughout the paper.

### 6.3.3 LeNet-5

The architecture of LeNet-5 is same as the one proposed in original LeNet [23] paper, with the only difference being that we change the final linear layer to output predictions for 100 classes since we are using CIFAR-100 dataset.

### 6.3.4 SmallCNN

SmallCNN is a simple CNN made of convolution, maxpooling and linear layers. The architecture starts with two pairs of convolution and pooling layers.Finally, we have a linear layer for prediction of class probabilities. The first and second convolution layers have 5 and 10 filters respectively with kernel size 3, stride 1 and same padding.

## 6.4 PermNet implementation & running time

The PermNets have been implemented in Pytorch. We made a few changes to depthwise convolution with multiple filters in Pytorch once we figured out that Pytorch performs channel expansion during depthwise convolution. We want the convolved output channels from each filter to be in the same order as filters themselves. Channel expansion performed by Pytorch goes against what we are trying to achieve in code. Hence, to fix that issue, we perform rearrangement of channels. This additional operation increases the training time of the network, especially if the number of channels to be rearranged is large. PermNets that use weighted channel summation do not use bias for their $FxF$, where $F > 1$ convolutions. However, PermNets using typical channel summation do use bias for all convolutions. PermNets generally take more time to train than baseline networks due to additional operations introduced. The non-deterministic PermNets with weighted summation take almost 3-5x training time when compared to baseline networks. The deterministic PermNets are a bit faster and take about 2x training time as compared to baseline.

## 6.5 Results

In this section, we provide the quantitative comparison between various PermNet variants and baseline architectures. We list the various fields that identify differences between architectures and their training procedure. These fields include learning rate (LR), optimizer used, learning rate scheduler usage, batch-normalization usage, permutation type and top-1 accuracy achieved.

For simplicity, in the permutation field, we would use square brackets to describe the number of channels we permute in each master layer and the master layers would be separated by a comma. On the other hand, for PermDeterNet and PermDeterWeightedNet, we would provide the number of additional constrained filters we generate for each master layer in curly brackets. The master layers are separated by comma. Inside the master layer, the permutation is applied in same manner for all of the convolution layers inside it.

The Table 6.1 showcases our emperical results.

| Model | LR | Optimizer | LRS | Batchnorm | Permutation | Accuracy |
|---|---|---|---|---|---|---|
| RResNet-18 | 0.001 | SGD | Y | Y | None | 64.38 |
| Weighted RResNet-18 | 0.001 | SGD | Y | Y | None | 63.7 |
| PermIterWeighted RResNet-18 | 0.001 | SGD | Y | Y | [0,1,1,1] | 58.79 |
| PermIterWeighted RResNet-18 | 0.001 | SGD | Y | Y | [0,2,2,2] | 60.36 |
| PermIterWeighted RResNet-18 | 0.001 | SGD | Y | Y | [0,5,5,5] | 59 |
| PermIterWeighted RResNet-18 | 0.001 | SGD | Y | Y | [0,10,10,10] | 59.62 |
| PermDeter RResNet-18 | 0.001 | SGD | Y | Y | {16,32,64,128} | 56.38 |
| PermDeter RResNet-18 | 0.0001 | SGD | Y | Y | {16,32,64,128} | 46.16 |
| PermDeter RResNet-18 | 0.001 | SGD | Y | N | {16,32,64,128} | 54.31 |
| PermDeter RResNet-18 | 0.001 | Adam | Y | Y | {16,32,64,128} | 61.43 |
| PermDeterWeighted RResNet-18 | 0.001 | SGD | Y | Y | {16,32,64,128} | 57.41 |
| PermAuto RResNet-18 | 0.001 | SGD | Y | Y | None | 57.14 |
| PermAutoMulti RResNet-18 | 0.001 | SGD | Y | Y | None | 66.01 |
| LeNet-5 | 0.001 | SGD | Y | Y | None | 41.7 |
| LeNet-5 | 0.001 | SGD | N | Y | None | 38.85 |
| Weighted LeNet-5 | 0.001 | SGD | Y | Y | None | 41.82 |
| PermDeter LeNet-5 | 0.001 | SGD | Y | Y | {6,16} | 42.86 |
| PermAuto LeNet-5 | 0.001 | SGD | Y | Y | None | 41.71 |
| SmallCNN | 0.001 | SGD | Y | Y | None | 36.99 |
| Weighted SmallCNN | 0.001 | SGD | Y | Y | None | 34.25 |
| PermDeter SmallCNN | 0.001 | SGD | Y | Y | {5,10} | 35.56 |
| PermAuto SmallCNN | 0.001 | SGD | Y | Y | None | 34.14 |
| ResNet-18 | 0.001 | SGD | Y | Y | None | 94.02 |
| PermShuffle ResNet-18 | 0.001 | SGD | Y | Y | [0,64,64,0] | 94.09 |
| PermShuffle ResNet-18 | 0.001 | SGD | Y | Y | [64,64,64,64] | 89.88 |
| PermShuffle ResNet-18 | 0.001 | SGD | Y | Y | [5,5,5,5] | 93.94 |
| PermShuffle ResNet-18 | 0.001 | SGD | Y | Y | [0,100,100,0] | 92.24 |
| PermShuffle ResNet-18 (transfer) | 0.001 | SGD | Y | Y | [5,5,5,5] | 93.5 |

Table 6.1: Performance of different models on CIFAR-100 dataset. The metric used for comparison is the top-1 class prediction accuracy.

# Chapter 7

# Findings and Recommendations

Upon, analysis of the results we can try to infer some general patterns that we see in permuted convolutions. The intuitive trend of having more permutations and training longer does not necessarily hold true in the experimentation. While the reason behind it is unclear, there are some general trends and findings that we do see through experiments that can lead to better training paradigm.

- Enabling permutations or shuffling in initial convolutional layers can have adverse effect on training. More experimentation is needed to visualize what features the PermNet learns in initial layers when permutations are turned on.
- Intuitively, batch normalization [24] can have adverse effect on PermNet training. However the results on PermDeter RResNet-18 indicate the otherwise.
- PermNets and WeightedNets generally take longer to train compared to other CNN counterparts due to additional constrained permutations or tensor rearrangement operations. The memory footprint is also larger due to creation of intermediate depthwise convolution output in each layer. PermShuffleNet is the only exception among these bunch in regard to memory usage. However its training time is slighly longer than that of the baseline network due to shuffling operation.
- We see that by permuting more filters for PermIterNet and PermShuffleNet variants, we do not see improved performance. In fact there is no general trend that

we see that could help us understand how number of permutations we consider affects the performance of the network.

- Some PermNets such as PermAutoMulti RResNet-18 achieve better accuracy than baseline architectures. However due to the tremendous increase in number of parameters of the network due to multiple 1x1 convolutions applied on depthwise convolved output, the gains in accuracy are deceiving and such networks are not practical.

- PermShuffleNet achieves very slight improvement over baseline networks, however the accuracy gain is insignificant compared to the additional training time we require from the network. There is also no generalized pattern regarding the number of filters we permute. Hence identifying the network that actually performs better than baseline requires large hyperparameter search grids, which are impractical.

- PermAutoNet achieves equivalent accuracy as baseline for some of the networks, but not for others. This is a surprising result that warrants further study. More hyperparameter tuning needed.

- From the results, we can conclude that none of the PermNets that we explore are significantly better than baseline. Hence we must try to find better PermNet architectures or try to utilize permuted convolutions in fundamentally different manner. We would discuss one such recommendation in future work.

# Chapter 8

# Future Work

In this section, we go over a few suggestions that we have for future work on permuted convolutions. We inferred from the results section that the PermNets that we we have explored do not give us satisfactory increase in accuracy as compared to baseline. One of the reasons can be that the additional generated filters fail to deliver on the promise of performing psuedo-width scaling.

However, the idea of permuted convolutions to generate additional filters may still be useful if we decide to flip the problem. With the current architectures, the additional filters are unable to help network learn better features. To alleviate this problem, we can take help from an expert network. With this intuition, we propose the following architecture as future work.

## 8.1 PermImitatorNet

Consider a network with width w for its convolution layer. This network has been trained optimally on the task at hand and hence is the expert network. Now consider a pre-trained version of such network that will be utilize to come up with optimal PermNet. The task here is to perform model compression using width reduction. The PermNet network we will consider will have lesser width than the expert network. We know that most CNNs are overparameterized and hence there is good scope for us to come up with ways to counter that inefficiency.

In this case, consider a PermNet that has half the width (w/2) as the expert network.

Let's call this new PermNet as PermImitatorNet, since its task would be to imitate the expert network. PermImitatorNet network is simply a PermDeterNet variant which has been subject to handling an additional loss function. In this case, this new loss is calculated by calculating the loss between the weights of additional constrained filters in PermImitatorNet and the extra w/2 filters in the expert network. By providing such additional loss to the network, we are forcing the additional constrained filters of the PermNet to learn similar patterns as extra filters in expert network. Hence, we are attempting to learn similar patterns as the expert network, however with half the number of parameters due to the usage of permuted convolutions.

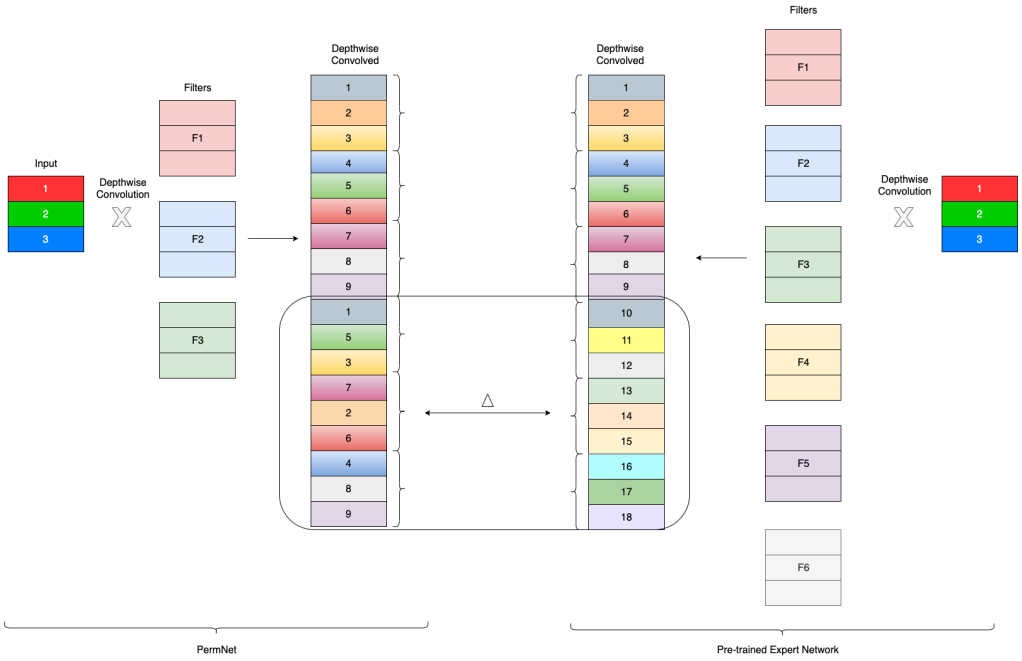


Figure 8.1: PermImitatorNet: The PermDeterNet attempts to imitate the expert network. This is achieved with the help of loss function added to the network that computes filter weights differences between additional constrained filters in PermNet and the extra filters in pre-trained baseline network.

PermImitatorNet would require extensive hyperparameter tuning and appropriate number of constrained filters for it to perform much better than similarly parameterized networks. Since such PermImitatorNets could possibly lead to extreme network

compression, this area deserves more research efforts.

# Chapter 9

# Conclusion

In this project, we explored how channel summation process in typical convolution introduces implicit bias that seems counter intuitive. We try to counter this bias with the help of the weighted summation process introduced in WeightedNet architecture. Then we explored how channel summation process in typical convolution can be generalized to allow permuted convolutions that attempt to facilitate information exchange among filters. We discussed how PermNets need to overcome both theoretical and implementation challenges in order for them to be practical. To this end, we discussed both stochastic and deterministic variants of PermNet. We performed extensive experimentation to compare PermNet with their typical CNN counterparts. We present our findings regarding the performance of PermNets. We discussed how PermNets struggle to beat baseline architectures, and even when they do beat baselines, the parameter costs or the training time increase makes them less viable for real world usage in their current formats. We then propose to flip the problem upside down by suggesting to focus on achieving network compression with PermNets in place of attempting to achieve accuracy gain with the variants. To this end, we suggest the extensive study of a new variant of PermNet called PermImitatorNet.

# References


[1] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 09–15 Jun 2019.

[2] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices, 2017, 1707.01083.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[4] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[5] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015, 1512.03385.

[8] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[9] Gao Huang, Zhuang Liu, Geoff Pleiss, Laurens Van Der Maaten, and Kilian Weinberger. Convolutional networks with dense connectivity. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[10] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017, 1704.04861.

[11] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019, 1801.04381.

[12] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size, 2016, 1602.07360.

[13] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1135–1143, 2015.

[14] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)*, 2016.

[15] Frederick Tung and Greg Mori. Clip-q: Deep network compression learning by in-parallel pruning-quantization. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7873–7882, 2018.

[16] Tianzhe Wang, Kuan Wang, Han Cai, Ji Lin, Zhijian Liu, Hanrui Wang, Yujun Lin, and Song Han. Apq: Joint search for network architecture, pruning and

quantization policy. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[17] Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, and Kyoung Mu Lee. Enhanced deep residual networks for single image super-resolution, 2017, 1707.02921.

[18] Hao Dong, Akara Supratak, Luo Mai, Fangde Liu, Axel Oehmichen, Simiao Yu, and Yike Guo. TensorLayer: A Versatile Library for Efficient Deep Learning Development. *ACM Multimedia*, 2017.

[19] Xintao Wang, Ke Yu, Shixiang Wu, Jinjin Gu, Yihao Liu, Chao Dong, Yu Qiao, and Chen Change Loy. Esrgan: Enhanced super-resolution generative adversarial networks. In *The European Conference on Computer Vision Workshops (ECCVW)*, September 2018.

[20] W. Yifan, F. Perazzi, B. McWilliams, A. Sorkine-Hornung, O Sorkine-Hornung, and C. Schroers. A fully progressive approach to single-image super-resolution. In *CVPR Workshops*, June 2018.

[21] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

[22] François Chollet. Xception: Deep learning with depthwise separable convolutions, 2017, 1610.02357.

[23] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[24] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.